SBIR N08-116
Phase 2 Final Report

29 February 2012

Prepared by
Object Computing, Inc. (OCI)
12140 Woodcrest Executive Drive, Suite 250
Saint Louis, MO 63141

Michael Martinez
Principal Investigator
Office: (314)579-0066
FAX: (314)579-0065
martinezm@ociweb.com

for

NAVAIR
E-2C Simulation Laboratory

Contract No. N68335-10-C-0043

Approved for public release; distribution is unlimited.

UNCLASSIFIED

Table of Contents

Drawing Index

1 Introduction

This document is the final summary report for Phase 2 of the Object Computing, Incorporated (OCI) research and development activities for SBIR N08-116 for the NAVAIR E-2C Systems Test & Evaluation Laboratory (ESTEL) (contract number N68335-08-C-0111). This report presents the technical objectives of OCI's activities during Phase 2, our findings and project activities, results from our activities and recommendations for further development, including plans for Phase 2.5 and Phase 3. The overall objective of this phase was to extend the feasibility study of Phase 1 and to illustrate the suitability for commercialization of the OpenDDS product [Ref 13.] for use in the simulation laboratory as middleware for the distribution of simulation data. During Phase 1 we found that this standards based Free Open Source Software (FOSS) product is appropriate for use in the laboratory environment. During Phase 2 we undertook a development process to prove the viability of the product.

During Phase 1 we identified areas where the OpenDDS middleware solution capabilities could be extended to meet perceived needs of the target laboratory as well as other government and non-government users. This included development and runtime environments as well as addressing stability during evolution of the middleware and the migration of supporting technologies such as the computing equipment, operating systems, development and runtime tools used by the solution. The Phase 2 activities implemented the highest priority extensions identified in Phase 1. Research during Phase 1 and collaboration with our customer identified the importance of a toolkit to use during software development as critical to use of this technology. This allows application developers and domain experts to focus on their problem domain rather than the details of middleware connectivity and usage. This software development kit (SDK) simplifies definition of the middleware segment of a system and allows low risk migration to next generation technologies. The Phase 2 activities also included implementation of features in the OpenDDS product to bring it into specification compliance. The underlying technology for transporting the data was simplified and improved during these activities. Some supporting tools for use during development and operation were also developed as part of this activity.

During the execution of this project, intermediate releases were made to the product's open source repositories [Ref 14.] and the implementations were tested not only as part of the SBIR but through use by other commercial projects. Results from this testing and usage were incorporated into the product as they were acquired. Training material for the SDK toolkit was developed and presented at the customer's laboratories as well.

2 Phase 2 Technical Objectives

OCI's Phase 2 proposal listed the following technical objectives for development:

1. Implement the Software Development Toolkit (SDK) identified during Phase 1.
2. Develop additional runtime support tools for the OpenDDS.
3. Develop implementation enhancements identified during Phase 1.
4. Execute performance characterization tests.

Our activities followed these objectives closely. As Phase 2 activities proceeded, the original objectives were extended to include additional areas of interest; the runtime support was extended to include creation of a spreadsheet add-in, and the implementation enhancements were extended to include implementation of the lower layer RTPS protocol specification [Ref. 9.] as an additional transport implementation and discovery mechanism.

3 Summary of Development

In this section, we briefly describe the results of our development activities with respect to each of the objectives listed in the section 2 *("Phase 2 Technical Objectives")*. Specific activities and results are presented in the section 4 *("Project Activities")*.

3.1 Development Support

The toolkit development included implementation of a UML graphical capture tool using the Eclipse platform. The specification meta-model was captured using the Eclipse Modeling Framework [Ref. 3.] Ecore meta-modeling facility, which is aligned with the OMG EMOF [Ref. 8.], using the OMG Platform Technical Committee UML Profile for DDS [Ref. 10.] as a starting point. During Phase 1, we identified a potential additional meta-model that was domain specific; but during Phase 2,

the code generation portion of the toolkit obviated the need for a separate domain specific profile to be incorporated into the toolkit. This simplified the middleware model capture and exposed the specification standard API directly to the applications. A training class including both lecture and laboratory exercises was developed as well to introduce the modeling toolkit to potential users.

3.2 Runtime Support

Runtime support included the development of a packet dissector plug-in for the open source Wireshark [Ref. 17.] network analysis tool. It also extended the ability to monitor ongoing service operation through the use of meta-data and published monitor topics. This activity was extended to include the creation of an Excel spreadsheet add-in.

3.3 Software Features

The implementation of OpenDDS was extended and tuned to bring it into compliance with the DDS specification [Ref. 7.]. This included implementation of the DCPS layer of the Object Model and Ownership, and Content-Subscription profiles. Internal tuning included publisher side content filtering, as well as an extended and improved IDL compiler, and a transport auto-selection mechanism. This activity was extended to implementation of the the lower layer RTPS protocol specification [Ref. 9.] as an additional transport implementation and discovery mechanism.

3.4 Testing

Performance tests developed during Phase 1 were executed on the updated releases of the OpenDDS middleware code base to ensure that Phase 2 development did not adversely impact performance.

3.5 Plans for Phase 2.5 and Phase 3

We have developed a plan for Phase 2.5 to continue extending the capabilities of the OpenDDS middleware solution. These plans include integrating OpenDDS with the FACE architecture interfaces [Ref 4.] and adding capabilities to implement the lower layer RTPS interoperability wire protocol. It

also includes implementing the higher layer DLRL abstraction layer from the DDS specification. The plans are described in section 5 *("Recommendations for Further Development")*.

4 Project Activities

In this section, we describe the specific activities and results of our research and development with respect to each of the objectives listed in section 2 *("Phase 2 Technical Objectives")*.

4.1 OpenDDS Software Development Kit (SDK)

The OpenDDS software development kit (SDK) is a graphical modeling toolkit that was identified during the Phase 1 research. During Phase 2, we implemented this toolkit [Ref 12.] to allow software developers to simply capture the middleware aspects of their applications. These models are stored in a way that can easily be used to migrate the middleware from current to future processing and development environments. The models are based on a meta-model that abstracts the important features of the middleware for an application. The meta-model is then mapped onto graphical components that allows models to be captured using a graphical rather than textual editor. The captured middleware models can then be used to generate code for the target development platform. The current target development platform is C++ language source code, along with supporting data definition files in OMG IDL format and build support files that can be used to specialize the build process for most commonly used development environments, including the GNU toolchain, Eclipse CDT, and Microsoft Visual Studio.

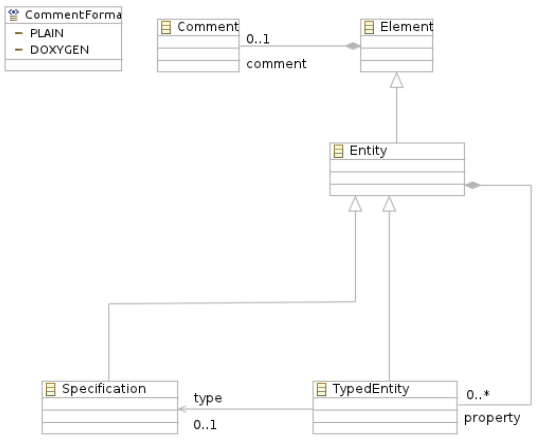
The meta-model and graphical capture toolkit are based on the Eclipse Modeling Framework (EMF) [Ref. 3.] and Graphical Modeling Project (GMP) [Ref. 2.] modeling frameworks. The code generation portions were developed using XSLT stylesheets to define transformations from the model content to the desired target formats. These transformations were then encapsulated for the user as an additional form based Eclipse plug-in that integrates with the graphical model capture portion of the toolkit. The toolkit was developed initially using the Eclipse 3.5 platform, and has been successfully executed for model definition and code generation using the 3.6 platform as well.

A training course was developed for the modeling toolkit. The training material recommends a work flow for capturing middleware models, tailoring the models for deployment, and generating the source code for compilation and linking with applications. The modeling toolkit can support other work flows tailored to specific projects as well, but initially we recommend use of the work flow defined by the training materials.

4.1.1 Meta-model

The elements of a middleware model were captured as a collection of coupled meta-models using the Eclipse EMF [Ref 3.] Ecore meta-model facility. This modeling approach is similar to the OMG eMOF [Ref. 8.] meta-model facility but is tailored specifically to the Eclipse environment. The initial meta-model was based on the draft OMG platform technology committee DDS Profile for DDS [Ref. 10.]. This meta-model was not sufficient to define the entire set of elements required in order to generate target middleware code. We extended and modified this profile to accommodate all of the known requirements for code generation. The meta-model consists of several distinct loosely coupled sections. These include the core, domain, DCPS, QoS, types, and topics meta-models. Each of these defines the relationship between various aspects of the middleware being described. Each of the leaf meta-classes in these meta-models will appear as element instances in the final middleware model captured by users. We publish these models in XML Schema Definition format to allow their use in validating XML instance documents for models. The schemas are located in the '*xsd*' folder of the *org.opendds.modeling.validation* plug-in provided as part of the modeling toolkit feature bundle.

The core meta-model section, shown in Drawing 1, defines "*Specification*" and "*TypedEntity*" classes that are used in other meta-models to distinguish between entities that have types associated with them and those that specify other information. The "*TypedEntity*" elements can have any number of properties and one or



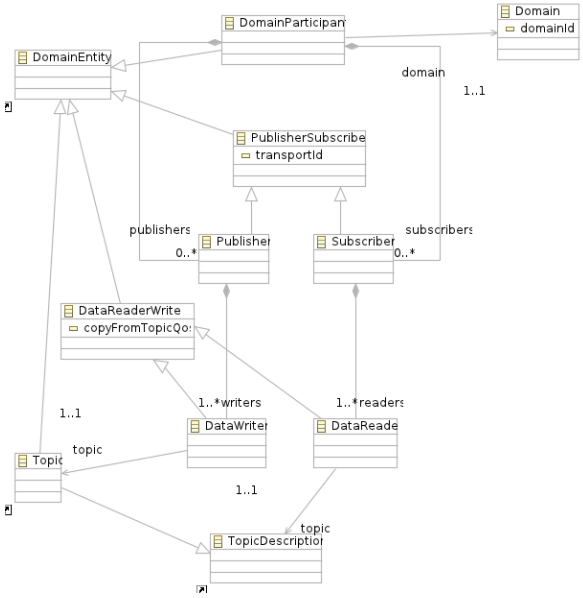
Drawing 1: Metamodel - Core

no specifications bound to them. These elements are extended by others in the subsequent meta-models and are not present in actual model documents themselves.

The Domain meta-model, shown in Drawing 2, defines the DDS "*DomainEntity*" and "*QosProperty*" elements that are used to define the actual Entities used by the middleware. These elements also are abstractions that are extended by other meta-models and do not appear directly in model instance documents.

Drawing 2: Metamodel - Domain

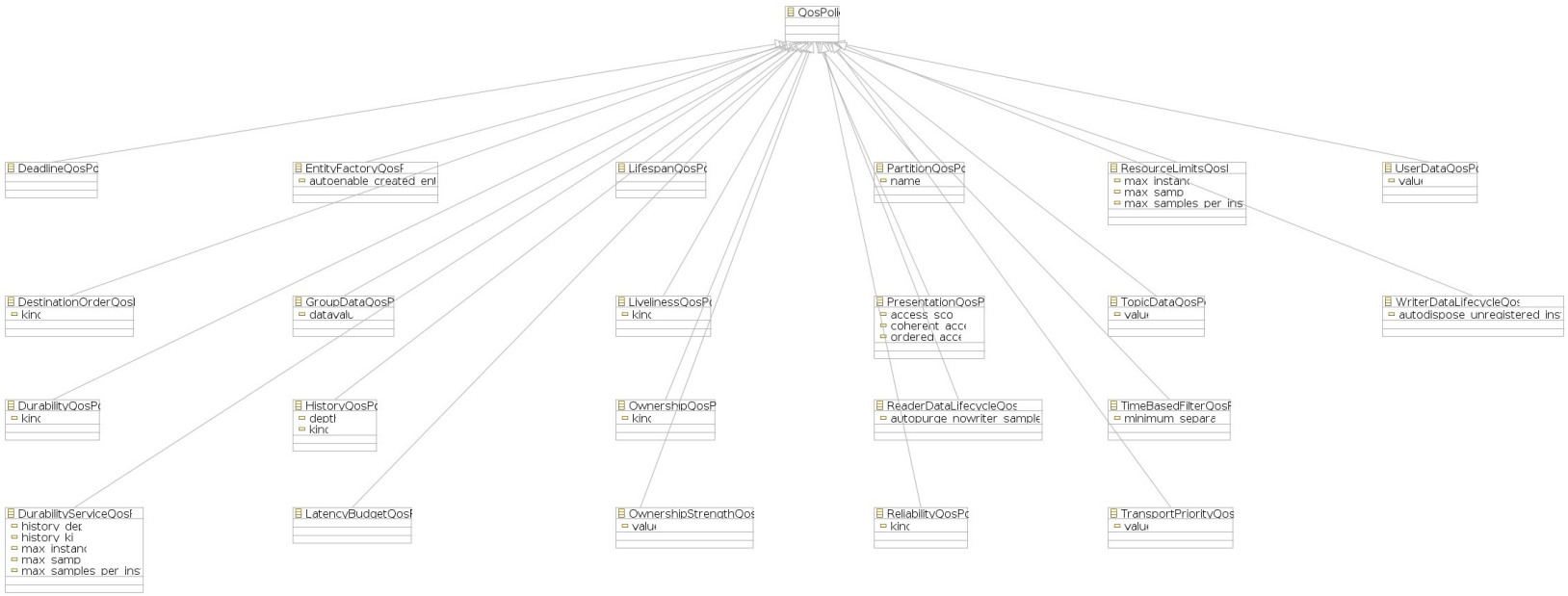
The DCPS meta-model, shown in Drawing 3, includes the major API Entities. This meta-model defines the elements that are most often used by applications to interact with the middleware. This meta-model includes external (references to other meta-model elements), abstract, and concrete elements. The elements corresponding to actual DDS Entities, such as "*Publisher*", "*Subscriber*", "*DataWriter*", "*DataReader*", "*Domain*", and "*DomainParticipant*" will be present in model instance documents.



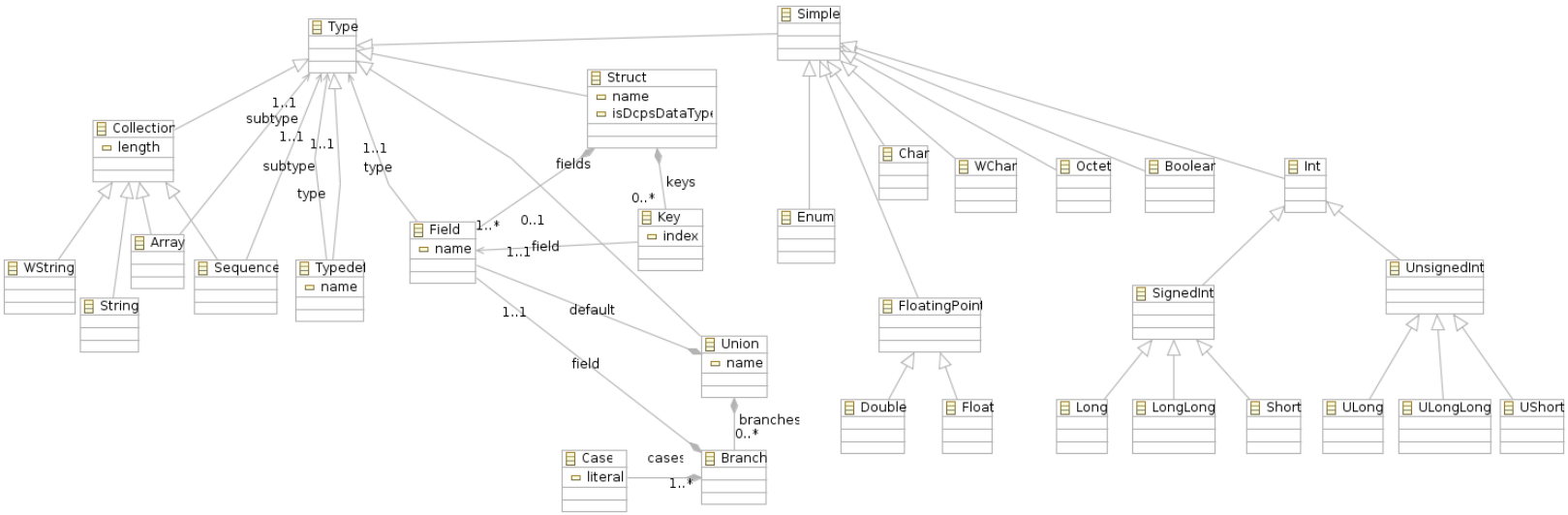
Drawing 3: Metamodel - DCPS

Quality of Service (QoS) Policies defined and used by the middleware are defined in the QoS meta-model, shown in Drawing 4, which simply instantiates each possible policy available to the middleware. The individual policy elements may be included in model instance documents.

Topics, which describe what data is transported in DDS applications, are defined in the Topic meta-model, shown in Drawing 6, where the different types of Topics are shown. The concrete model elements include basic "*Topic*", "*MultiTopic*", and "*ContentFilteredTopics*" as well as the "*TopicDescription*" used by receive side processing. The binding to data descriptions is included by referencing the "*Struct*" from the Types Meta-model.



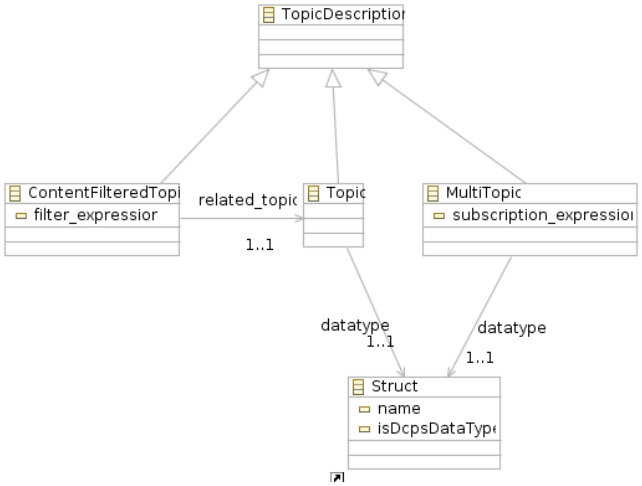
Drawing 4: Metamodel - QoS Policies



Drawing 5: Metamodel - Data Types

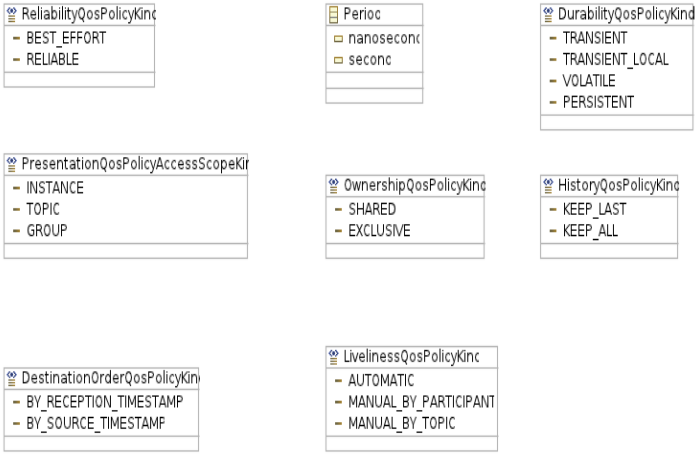
The data type definition meta-model is in two parts. The data structures themselves are shown in Drawing 5, with an additional meta-model defining information for enumerations included in Drawing 7.

The types include the possible simple and complex types that can be transported over DDS. This includes structures (the only type that can be bound to a "*Topic*" for transporting data) and collections. The ability to alias type names using the OMG Interface Definition Language (IDL) typedef mechanism is allowed as well. The enumerations meta-model includes specific enumerations and values that are used by the DDS middleware.



Drawing 6: Metamodel - Topics

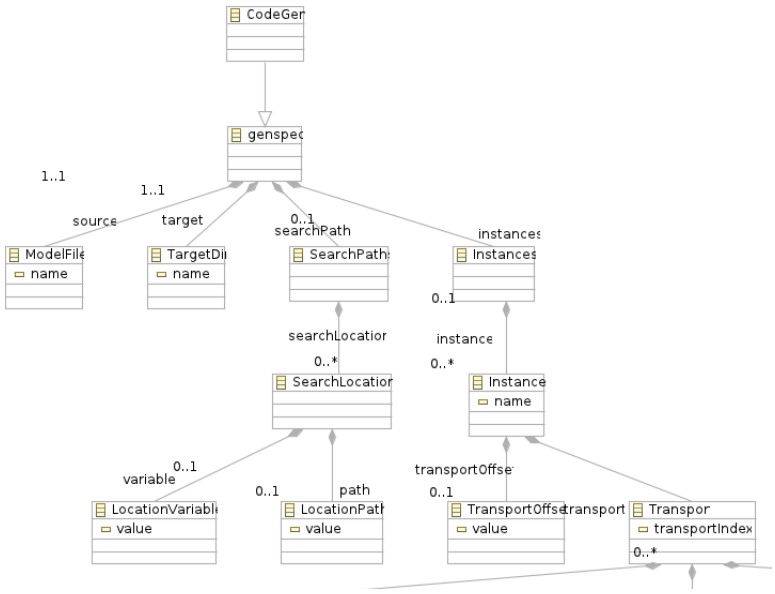
The Application meta-model defined by the proposed UML Profile for DDS [Ref 10.] was omitted from this collection of meta-models since the binding of the middleware model with specific application features is considered to be part of deployment and not of middleware definition. This allows the resulting toolkit to defer inclusion of the middleware into applications until the actual applications have been better defined. This also allows the same middleware model to be used in more than one context or to be shared or used more than once within the same system. It also facilitates the model being migrated, along with an application, by modifying only the deployment information rather than the model contents. It also facilitates migration of the model and application



Drawing 7: Metamodel - Enumerations

through the modification of only the deployment information, without any changes to the model contents.

Additional elements to help in code generation and deployment are defined in the generator meta-model. This meta-model defines elements for one or more instances of the model to be bound to applications. Additional elements are then included that define the build environment, target application and location, and transport configuration parameters. A subset of this model is shown in Drawing 8.



Drawing 8: Metamodel - Deployment

The model contains additional detail for transport configuration as well. Each transport includes several specific elements.

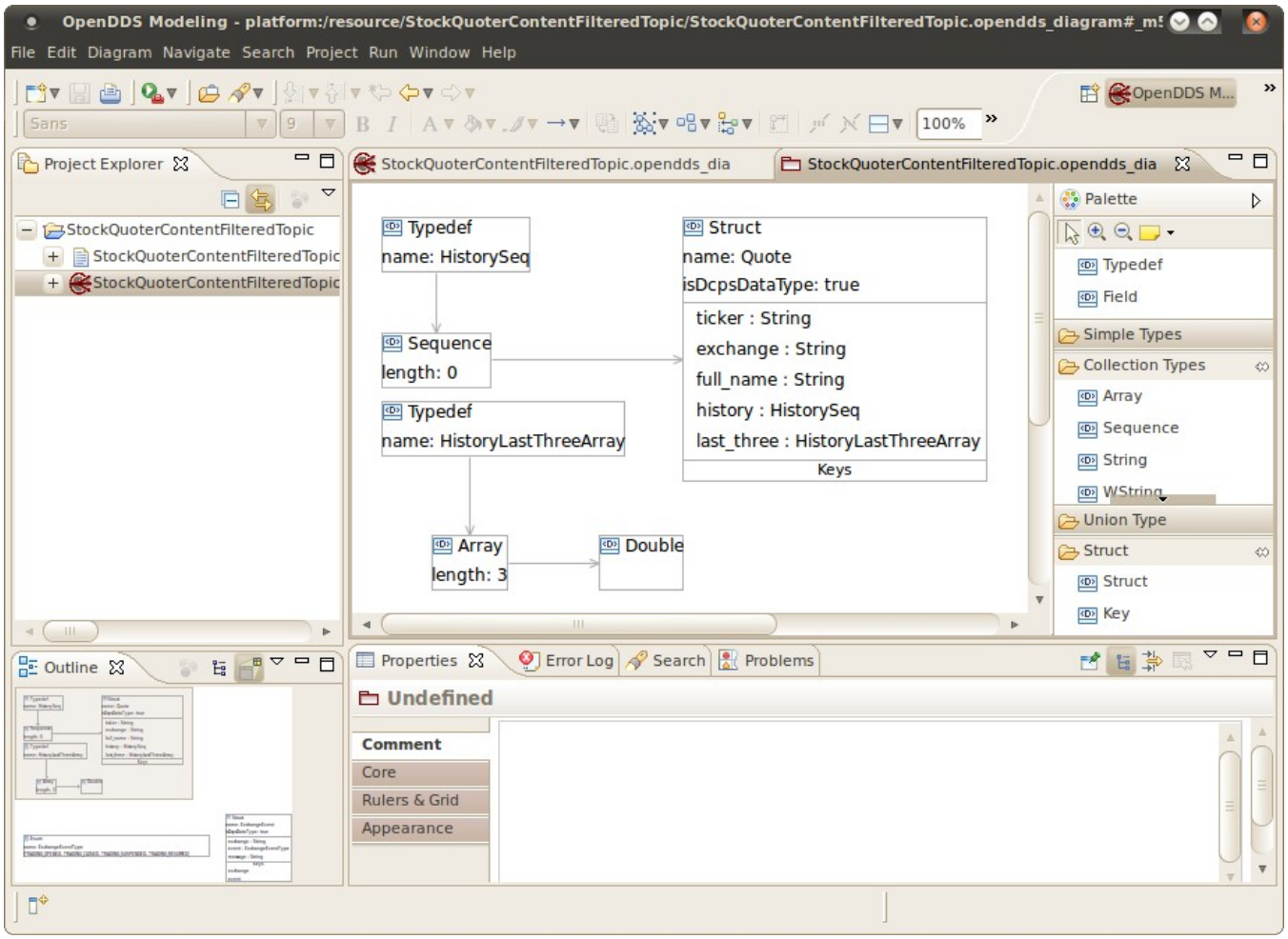
4.1.2 Model Capture

Middleware information is captured using the graphical model capture facility provided by the Eclipse GMP [Ref. 2.] as well as using form based information capture. Various diagrams and forms are combined to form the modeling toolkit. These include a top level package diagram, structural diagrams defining the service topology, policy diagrams defining Quality of Service policy values, data definition diagrams, build environment configuration form, deployment definition form, and a code generation specification form that also includes controls to manage the code generation process. The GMP diagrams are bound to the EMF meta-model to define the model elements.

The data, structure, and policy diagrams can be shared by multiple resources within the model or across models. This provides flexibility in the creation and management of models for a given system

or group of systems. Data can be defined across single or multiple systems. Policies can be defined with a broader scope than just a single application. Models of the middleware structure can be factored into reusable pieces that can be combined at system definition time to provide a quick middleware solution to application requirements.

The modeling toolkit is implemented as a feature bundle, or collection of plug-ins, for the Eclipse software development platform. The feature is made available via the standard installation and update mechanisms provided by Eclipse [Ref. 12.]. When started, the overall application and its perspective appears as in Drawing 9.

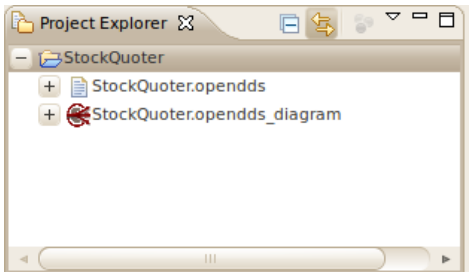


Drawing 9: Eclipse Software Development Kit (SDK) Feature

The project explorer window shows a tree view of the workspace that will contain both the model and the generated code. The editor window will display the currently active diagram as a canvas and

palette that allow the user to capture the model. It will also display the code generation forms that are used to tailor the build process and perform the generation steps. The outline, properties, problems, and error log views are used consistently with standard use of the Eclipse IDE.

Diagrams are stored in the Eclipse workspace as two linked XML files. Drawing 10 shows the two files in the project explorer view that store a single model. The file with the '*.opendds*' extension contains the model elements stored in XML format using a schema defined by the toolkit meta-model described in section 4.1.1 *("Meta-model")*. The file with the '*.opendds_diagram*' extension contains XMI (XML Metadata Interchange) format [Ref. 11.] data that represents the graphical diagram information, with external references to the XML file with the model data. Opening this file will invoke the main diagram editor for the model.

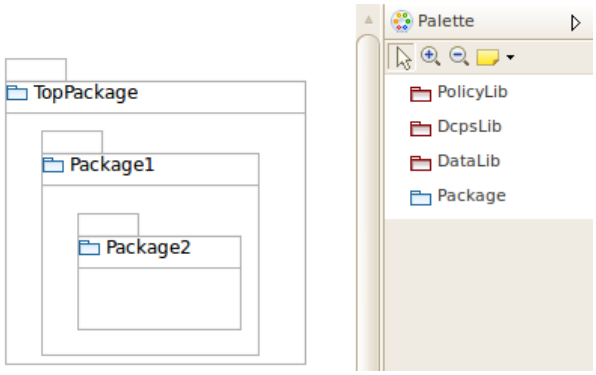


Drawing 10: Model Capture - files

The toolkit has the capability to generate a default graphical model file by reading a model definition XML file. This can be used to import external models, such as legacy applications, that can be translated to the XSD schema for the meta-model. Imported models can then inter-operate with other models within the toolkit.

4.1.2.1 Main Diagram Editor

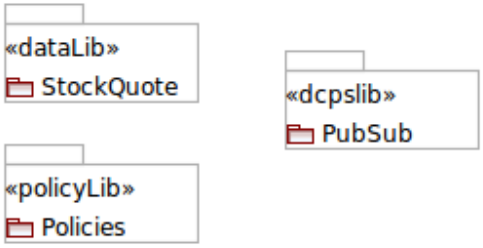
After a model has been created, the first diagram used to capture the middleware is the top level main diagram. This is a UML package diagram and will contain both organizational as well as functional model element packages. Drawing 11 shows the package diagram palette, which includes the leaf node stereotype elements as well as basic packages which can be used for containment. It also shows a hierarchy of basic packages illustrating containment. Containment in this diagram will



Drawing 11: Main Drawing - Package Hierarchy

translate into C++ classes in the generated code, which allows models to match and integrate with existing legacy code.

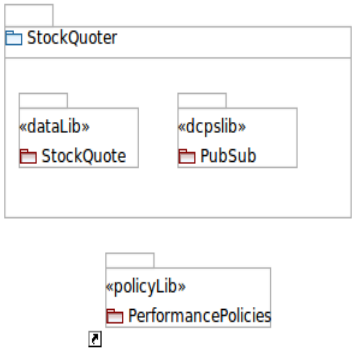
A model will contain library package types corresponding to sub-models that can be DCPS model elements, data definitions, and QoS policy definitions. These sub-models, which are leaf packages in the diagram, are indicated by guillemots («·») with the model stereotype indicated. An example of these leaf packages is shown in Drawing 12.



Drawing 12: Main Drawing – Sub-models

The «*dataLib*» stereotype is used for data definition model elements, the «dcpslib» stereotype is used for DCPS middleware model elements, and the «policyLib» stereotype is used to define common Quality of Service (QoS) policy values for use in the model. Each of these leaf packages will open a separate editor when selected.

Packages can be imported from external model definitions as well. Imported packages will have a reference decorator (a boxed arrow at the lower left corner) to visually indicate to the modeler that this is an external reference. This is shown in Drawing 13. External model references allow models to be composed of previously defined models and to share common definitions across a project. Typically the data definitions or common policy values would be defined once and used throughout a project to ensure common definitions are consistent throughout the project and all constituent models.

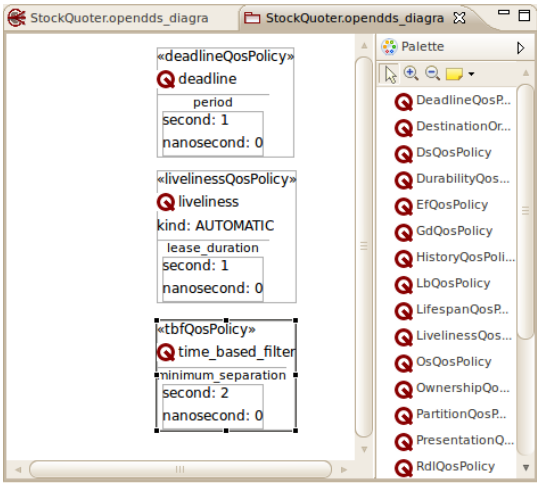


Drawing 13: Main Diagram - External Reference

Selecting a local leaf package and activating it in the main diagram will open another editor session with an editor specific for the type of elements of that leaf package. Each of the DCPS, data, and policy definition models have a dedicated editor for that diagram type. This simplifies model capture by restricting the available palette for adding elements to a diagram to those appropriate for that diagram.

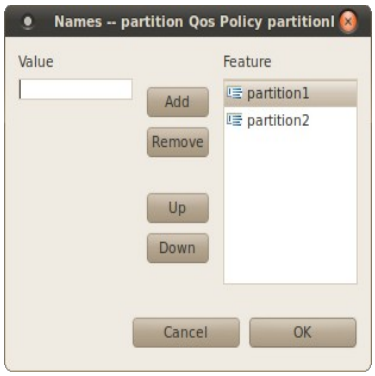
4.1.2.2 QoS Policy Model Editor

The QoS policy model editor allows the capture of policy values that can be shared across any model that contains the definitions, either directly or by reference. The QoS policy editor has a palette containing all allowed policies that can be added to the editor canvas to make them available to other model elements. The editor is shown in Drawing 14. The defined policy elements are named in the model and then incorporated into other model elements by referring to the qualified name of the policy.

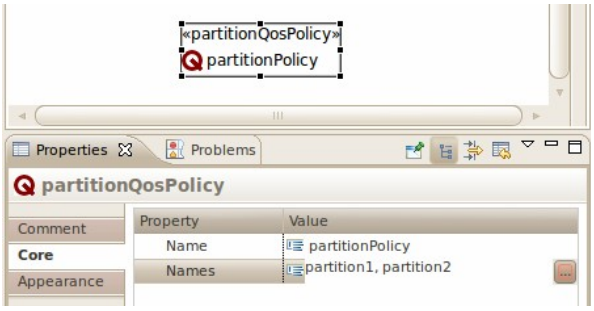


Drawing 14: Policy Diagram

Most policy values can be entered directly into the drawing elements in the model. Some are more complex and will activate a dialog to edit the values. The PARTITION policy has a dialog for entering and modifying the values as shown in Drawing 15. The dialog can be activated by selecting the corresponding value in a «*partitionQosPolicy*» model element Properties View as shown in Drawing 16.



Drawing 15: Policy Diagram -- «*partitionQosPolicy*» dialog

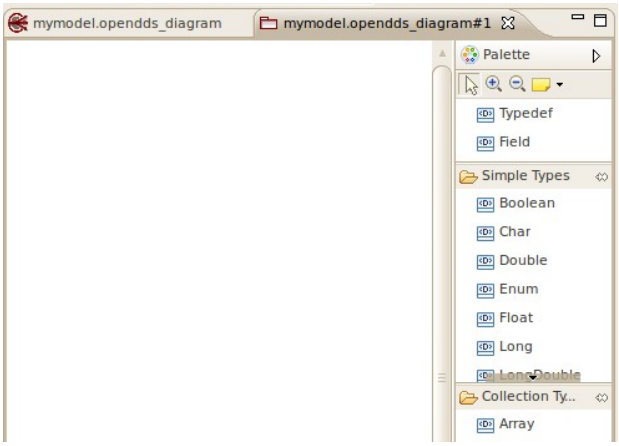


Drawing 16: Policy Diagram - Properties View

There are no additional relationships that can be defined in this editor; it simply contains an element for each policy that can be shared. The policy values defined in these elements are incorporated into the generated code by value. That means that there is no separate code artifact generated corresponding to these elements.

4.1.2.3 Data Definition Editor

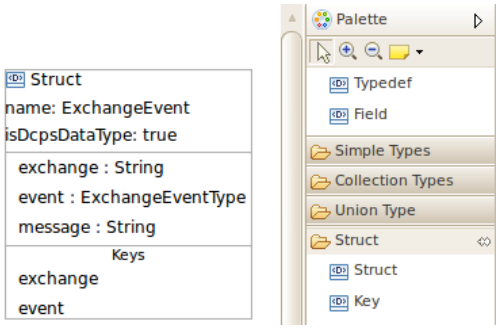
The data definition editor allows the modeler to capture data definitions used by the middleware. These data definitions are constrained to define only data that can be transported by the DDS service. The model will be used to generate CORBA Interface Definition Language (IDL) source code that can then be compiled to the desired target language supporting code.



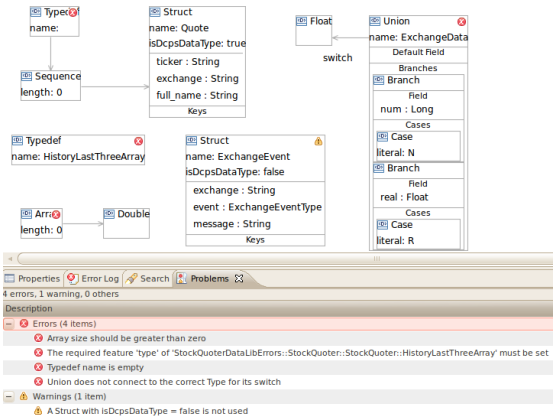
Drawing 17: Data Definition

Drawing 17 shows the data definition editor palette and canvas. The palette contains sections for simple types, collections, unions and structures as well as elements defining fields in a structure and *typedef* elements that can be used to alias the same definition with different names.

Structure elements, shown in Drawing 18, have containers for defining attributes, fields, and keys. The '*isDcpsDatatype*' attribute is used to control the generation of additional type support code specific to DDS and is required for structures that will be bound directly to a topic. Structures that are only contained by others and not bound to topics directly can suppress generation of this additional code.



Drawing 18: Data Definition - structures



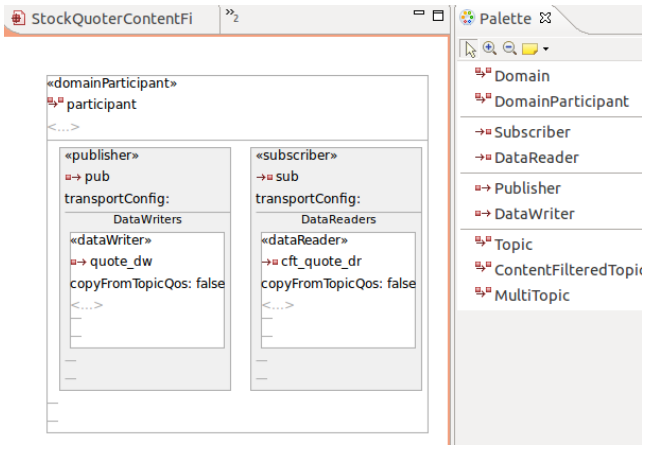
Drawing 19: Data Definition - validation

In addition to defining data types, aliases, and relationships between types, the data definition editor also validates the captured model elements to ensure consistent and complete generated target code. Drawing 19 shows several data type elements specified along with the '*Problems*' Eclipse view with error and warning

reports for those elements. The standard Eclipse IDE process flow can be followed to track and correct the reported errors.

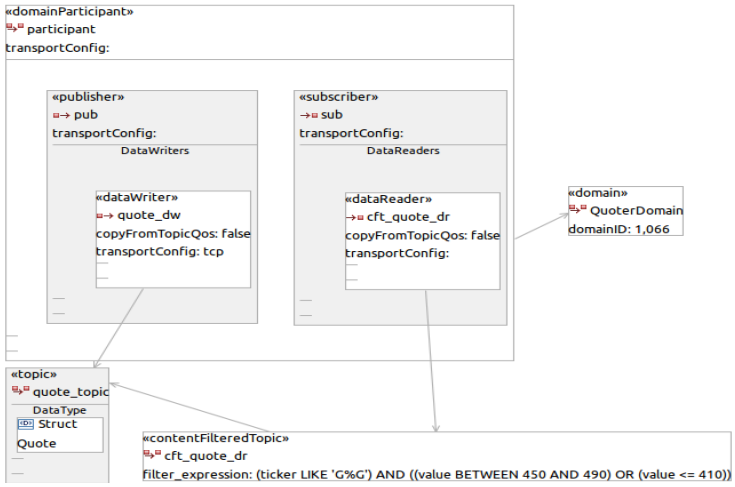
4.1.2.4 DCPS Model Editor

The DCPS model editor brings all of the other model elements together and defines the actual middleware structures to be used by applications. The DCPS elements include the *Domain*, *DomainParticipant*, *Publisher*, *Subscriber*, *DataWriter*, *DataReader*, and various *Topic* classes. The application uses these objects to interact with the DDS API. Drawing 20 shows a DCPS model editor open with the palette and a canvas with a *DomainParticipant* with a *Publisher* with *DataWriter* and a *Subscriber* with *DataReader* defined. The palette contains the elements that can be created in the canvas directly. In addition, shared elements can be included in the model by using the project view to navigate to the external elements and adding them directly to the canvas. This is typically done with *Topic* elements, which are shared between various applications in order to distribute data.



Drawing 20: DCPS Model Editor

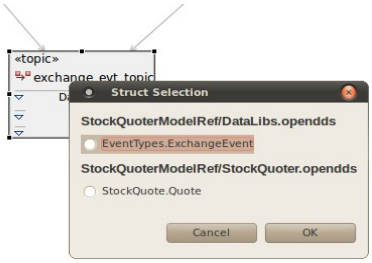
The elements captured in the DCPS editor include containment relationships and associations that are represented graphically. A partial model is shown in Drawing 21 that includes a *Publisher* and *Subscriber* contained within a *DomainParticipant*, and a *DataWriter* contained by the *Publisher* and a *DataReader* contained by the *Subscriber*. Associations between a *Domain* and *DomainParticipant* are



Drawing 21: DCPS Model Editor - element relationships

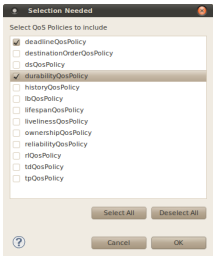
shown. Associations between *Topic* and *ContentFilteredTopic* and the corresponding *DataWriter* and *DataReader* are also shown. Attributes on the elements define additional aspects of the model, such as the *Topic* data type and filter expressions.

Data types are bound to *Topic* elements by using a dialog selection that provides all of the available data types that can be used, including those from externally referenced models. Drawing 22 shows data types that are available from two different models that can be bound to the selected *Topic*.

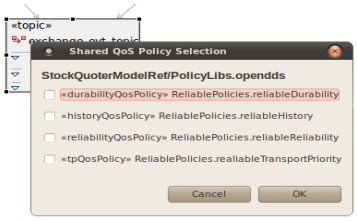


Drawing 22: DCPS Model Editor - data type selection

In addition to data types, each element of the DCPS model can have various QoS policies defined for them. The DDS specification defines policies for each of the elements. The modeling toolkit allows the policies appropriate for an element to be attached to and values assigned each element. If a policy for an element is not attached, the default value will be used by the generated code. Policies are attached by value. If the values are not from a shared QoS Policy model element, they are private to the element and are called "custom" policies. They can be added to any element through the use of a dialog as in Drawing 23. It is also possible to assign policy values using elements defined in a local or external model by the dialog shown in Drawing 24.



Drawing 23: DCPS Model Editor - QoS policy selections

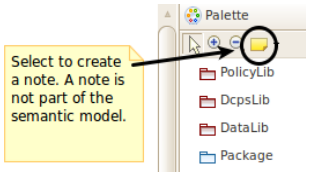


Drawing 24: DCPS Model Editor - shared policy values

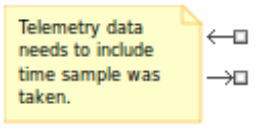
4.1.2.5 Annotations

Each of the diagrams allows annotations to be added to both the diagram and the generated code. Annotations to the diagram are not propagated to the generated code, but remain on the diagram for the modeler to provide additional model information to the reader. Drawing 25 shows the palette selection

for adding a note to a diagram, and Drawing 26 shows a note with its connection handles. These handles can be used to connect the annotation to diagram elements.



Drawing 25: Note palette selection

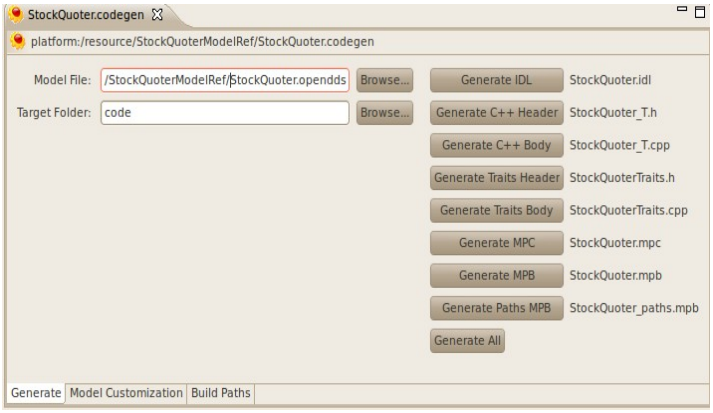


Drawing 26: Note connection handles

Annotations to the model elements can include comments that are added to the generated code. These comments can be in standard native format or in a *doxygen* [Ref 1.] style format for use in post processing the code to generate automatic documentation. This is done by using the Properties view of a selected element and entering a single or multi-line comment in the comment property. Annotations to DCPS model elements will appear in the internal generated C++ code headers. Annotations to Data Definition model elements will appear in the generated IDL code that is then compiled into the C++ source code. Typically only the data definition comments would be useful to a project; the DCPS element comments appear in generated code that is not normally viewed by the application developers.

4.1.3 Code Generation

Once the middleware semantic model for a system has been captured, it can be used to generate code to one or more destination directories. The generated code consists of OMG IDL data definitions that can be compiled into a target language, C++ source code that can be compiled and linked into applications, and build support files that can be used to create makefiles or project files used to build and link applications for a system. This code generation is done using a form editor that includes a *Generate* tab, as shown in Drawing 27, where the target directory and source model file can be selected. The code generation specification is stored in a separate XML file



Drawing 27: Code Generation - generate tab

with a separate schema from that of the middleware models. This allows the same middleware model to be targeted to different build directories with different code generation constraints, which enables reuse of common models within a system.

The Target Folder of the code generation specification is where all generated code will be placed. The recommended work flow places this directory as a sub-directory of an application that will be linking the source model. Two styles of model partitioning can be used with the toolkit – one where each model is associated with a single application of a system and includes data types and topics by reference from centralized definition models. This reduces the amount of code linked to each application. Another, equally valid, style of modeling includes defining the entire system middleware as a single model (or set of cross referenced models) that is linked to all applications within a system. Partitioning models into individual sets of elements for use by a system is done to support the needs of the target system and can combine use of these two styles of organization.

The *Model File* field of the code generation specification should refer to a file that is created by the graphical model editors described in section 4.1.2 (“*Model Capture*”). It will be used to provide the middleware semantic model for generating code. External model elements, included by reference into the model, will be included by reference in the generated code as well.

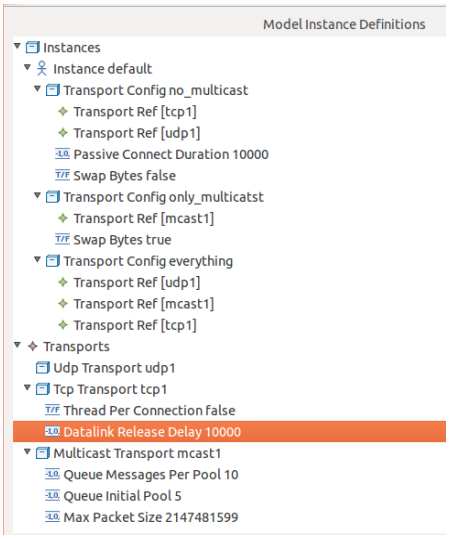
The source code generated into each separate target directory will be compiled into a single link library (typically a dynamically linkable library) that is then linked with applications that require those model elements. A single model can be used to generate tailored code into different target directories, which will then be used to build separate libraries linkable by applications. This one to many mapping of middleware models to link libraries means that designing the system middleware should account for what middleware elements are needed by applications and partitioning the models accordingly.

Code generation is performed by using an XSLT stylesheet for each target file type to transform the semantic model elements of the source model file or code generation specification into the desired format. This can be done directly from the editor or scripted using jar files containing the stylesheets and XSLT transformation engine.

4.1.3.1 Model Customization

The code generation specification includes the ability to customize the generated code by specializing the model that is generated. This is done using the *Model Customization* tab of the code generation editor. This tab allows the specification of transport configuration details and binding those to semantic elements of the source model. This is done using a tree editor with two root elements as shown in Drawing 28.

The *Instances* root element of the tree allows customization of the model elements in more than one way within the same generated code. As an example, if a model segment contained definition for a computation element that could be replicated several times within a single application, the model could be captured once, and then instantiated in the application multiple times as separate instances. A single default instance is available when a specification is started, and the user can then create more instance definitions as required. The customization of instances includes the definition of *Transport Config* lists that are used by the model. The model can bind these configuration lists to elements of the model as a property of the elements in the DCPS model editor. The lists then reference one or more specific *Transport* definitions as well as some common properties for all transports.

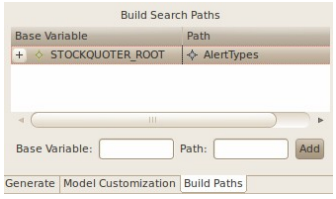


Drawing 28: Code Generation - Model Customization tab

The *Transports* root element of the tree allows definition of individual transport instances. Each of these transport definitions includes any non-default property values for the transport. The instances are included in the *Transport Config* lists defined for the *Instances* above to allow the specification of transport behavior for a particular model.

4.1.3.2 Build Support

Build support files are generated in addition to the C++ source code and IDL data definitions. These build support files are in the form of



Drawing 29: Code Generation - Build Paths tab

project definition files for the Makefile, Project, and Workspace creator (MPC) [Ref. 6.] tool that can be used to generate GNU makefiles and Visual Studio project and solution files. MPC supports other build systems as well. The MPC input files generated include specification of directories to search for include files and libraries during building as well as defining dependencies between generated model libraries. The Build Paths tab of the code generation editor, shown in Drawing 29, allows for the specification of variables and paths, both absolute and relative, that will be searched at build time for include files and link libraries. These can be specified relative to environment variables to allow deferring the actual location specification until the build is actually performed.

4.1.4 Application Integration

Once the semantic content for middleware has been captured and the model customized and code generated, it can be linked in to applications for use. This is done through the use of a support library specific to the generated code. This simplifies lifetime management for middleware Entities and provides for consistent access to all model elements.

```
// Generated Model include file
#include "../model/SatelliteTraits.h"

// Support Library include file
#include <model/Sync.h>

int main(int argc, char* argv[])
{
  try {
    // Using support library
    OpenDDS::Model::Application app(argc, argv);

    // Using generated model code
    Satellite::PubSub::DefaultSatelliteType model(app, argc, argv);
    using OpenDDS::Model::Satellite::PubSub::Elements;
    DDS::DataWriter_var writer =
      model.writer(Elements::DataWriters::telemetry_base_dw);
```

Code 1: Application using generated code

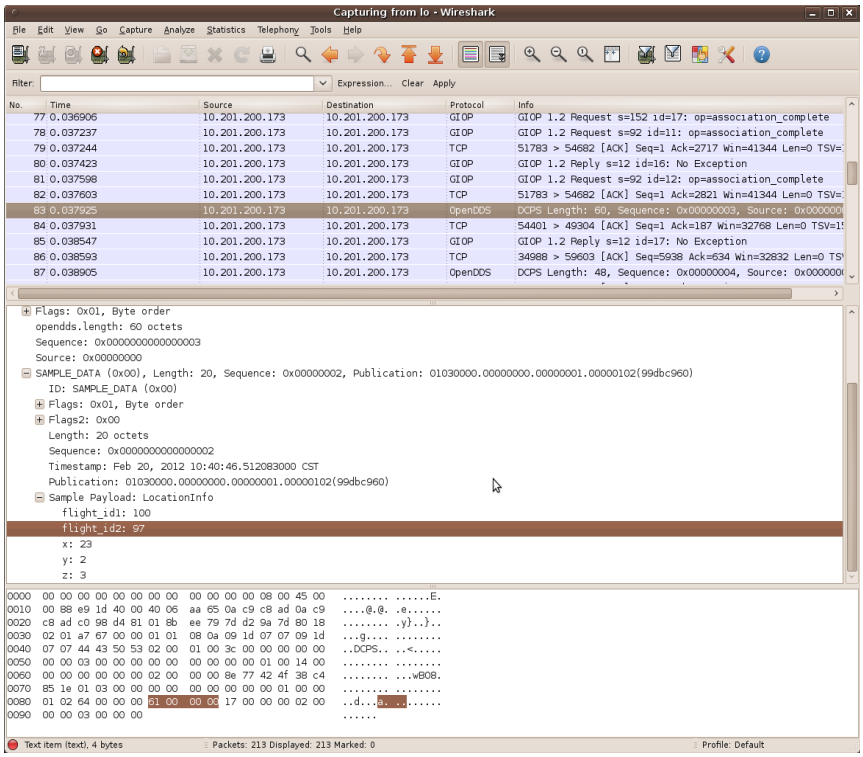
An example fragment of application code incorporating model elements is shown in Code 1 which demonstrates the inclusion of both the generated model and support library header files. It also shows the instantiation of the support library as an object (*app*) and the middleware model as an object (*model*). Access to DDS API Entities is then performed via accessor methods that use enumeration values to select the desired Entities by type. Use of the enumeration values allows efficient storage of the Entities and minimizes access times. Once accessed, the DDS Entities are used directly as standard Entities (which they are). Use of the object representations makes the lifetime of the middleware and the support library explicit in the application and ensures that both will be shutdown correctly after use.

4.2 Runtime support tools

Additional runtime support tools were developed to simplify the use and tuning of OpenDDS in systems. A Wireshark [Ref. 17.] dissector plug-in was developed to allow network traffic to be captured and analyzed during system operation. A DDS monitoring service was developed that subscribes to the DDS metadata and additional monitoring Topics to display operational status information.

4.2.1 Wireshark Dissector

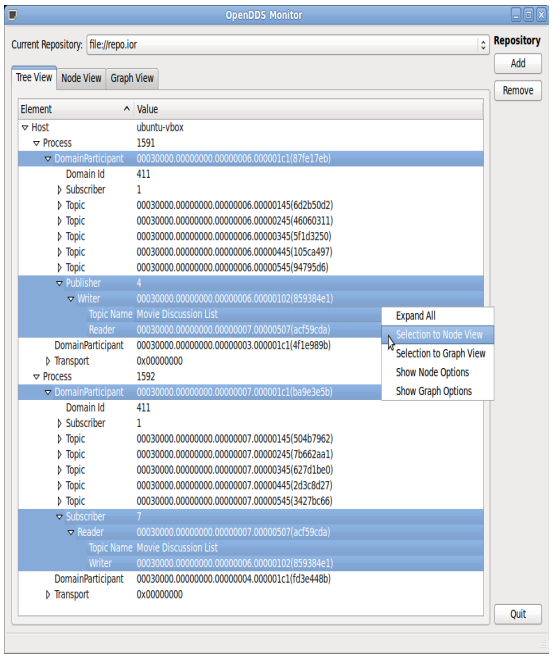
Wireshark is a ubiquitous packet analyzer that allows network traffic to be filtered and parsed. It parses packets using plug-ins that understand the different protocols involved. These plug-ins are called dissectors. We developed a dissector for the transport layer and sample headers of OpenDDS traffic. We also included the ability to link in information about user defined data types so that they can be dissected as well. This dissector parses the OpenDDS specific transport packets over either UDP or TCP transports and displays the transport header, sample header, and sample contents. The IDL compiler was extended to generate parsing support that can be used by the dissector to understand and display the sample data. Drawing 30 shows a Wireshark session including an OpenDDS sample with the user defined data dissected and displayed.



Drawing 30: OpenDDS Wireshark *Dissector*

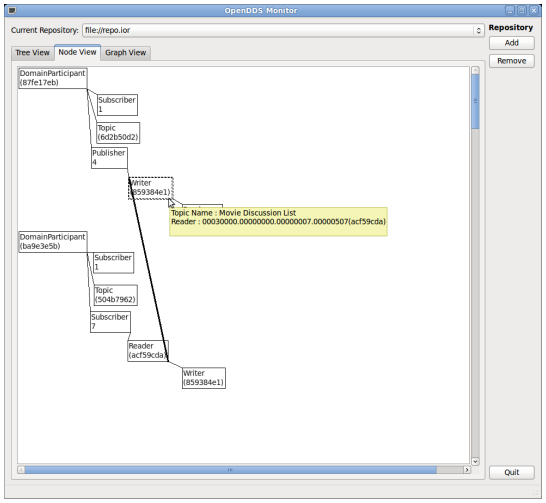
4.2.2 Service Monitor

A monitoring facility was developed that reads and displays DDS service metadata as defined by the specification along with additional status topics defined to expose internal operational state. This facility is made available as both a standalone Qt based GUI application as well as an Excel spreadsheet add-on.

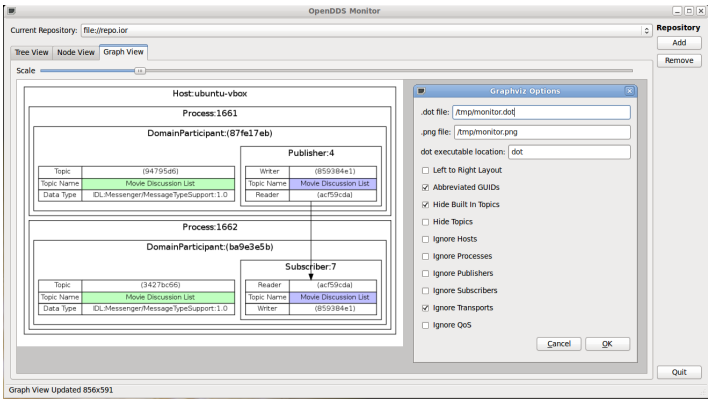


Drawing 31: OpenDDS Monitor - Qt GUI Tree View

The standalone application includes three views: Tree, Node, and Graph. The graphical Node and Graph views are generated from the data using the GraphViz [Ref. 5.] applications to automatically generate a graphical view from dependency data. The Tree view shows the entire data set as a navigable tree. The values of tree nodes are displayed as the information associated with that node. For example, in the Tree view of Drawing 31, the DDS Entities have displayed values corresponding to their GUID values within the service. They are shown as expandable elements with the sub-elements representing contained



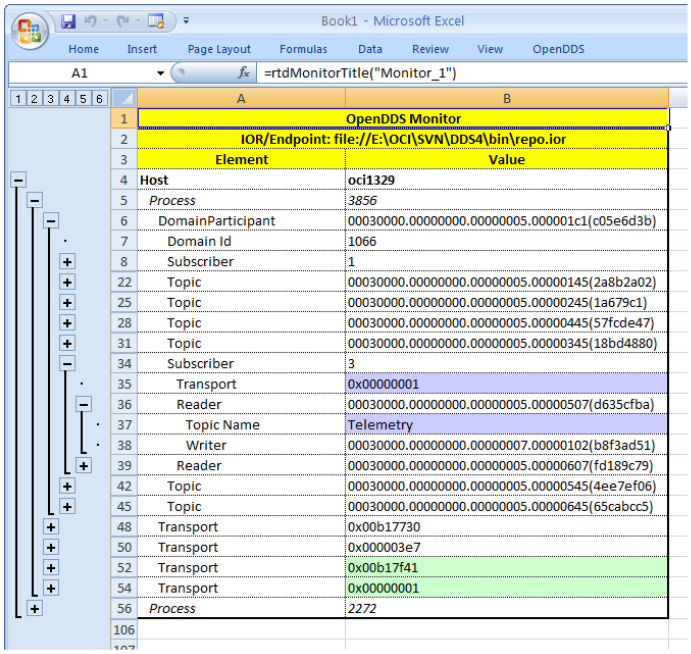
Drawing 32: OpenDDS Monitor - Qt GUI Node View



Drawing 33: OpenDDS Monitor - Qt GUI Graph View

nodes, such as the DomainID value or Subscriber node elements contained by a DomainParticipant node. The Node View of Drawing 32 shows the same information, but as a set of connected nodes, with cross references between elements shown as arcs. This can be seen with the relationship of the Reader and Writer association in the diagram. The Graph View shown in Drawing 33 illustrates the same elements in a graphical format and also includes association between elements, as the association between the Reader and Writer for the "*Movie Discussion List*" Topic in the diagram. The diagram also includes a dialog that can be used to restrict the amount of information in the display, which is useful for large systems and to suppress repetitive and common information such as the Builtin Topics defined by the specification.

The data displayed joins information provided by the DDS Built-in Topic metadata and monitor data that has been defined specifically for this feature. The monitor data includes both static information about the elements of the service as well as dynamic information that provides a snapshot of current operational state. These topics are published by linking applications with a separate dynamically linked library and enabling the feature, which allows applications to avoid any cost associated with the feature other than the DDS defined metadata. The display applications generate a view of the available data, and do not require all possible data to be available for display.



	A	B
1	OpenDDS Monitor	
2	IOR/Endpoint: file://E:\OCI\SVN\DDS4\bin\repo.ior	
3	Element	Value
4	Host	oci1329
5	*Process*	*3856*
6	DomainParticipant	00030000.00000000.00000005.000001c1(c05e6d3b)
7	Domain Id	1066
8	Subscriber	1
22	Topic	00030000.00000000.00000005.00000145(2a8b2a02)
25	Topic	00030000.00000000.00000005.00000245(1a679c1)
28	Topic	00030000.00000000.00000005.00000445(57fcde47)
31	Topic	00030000.00000000.00000005.00000345(18bd4880)
34	Subscriber	3
35	Transport	0x00000001
36	Reader	00030000.00000000.00000005.00000507(d635cfba)
37	Topic Name	Telemetry
38	Writer	00030000.00000000.00000007.00000102(b8f3ad51)
39	Reader	00030000.00000000.00000005.00000607(fd189c79)
42	Topic	00030000.00000000.00000005.00000545(4ee7ef06)
45	Topic	00030000.00000000.00000005.00000645(65cabcc5)
48	Transport	0x00b17730
50	Transport	0x000003e7
52	Transport	0x00b17f41
54	Transport	0x00000001
56	*Process*	*2272*
106		

Drawing 34: OpenDDS Monitor - Excel Addon

The Excel spreadsheet addon displays the same data from within a spreadsheet as shown in Drawing 34.

4.3 Implementation Enhancements

The OpenDDS free open source software (FOSS) implementation of the DDS specification implemented the minimum and persistence profiles of the specification at the start of Phase 2. During Phase 1 we identified development needed in order to fully comply with the DCPS portion of the specification. This included the addition of the DCPS layer of the object model profile, the ownership profile, and the content-subscription profile. These were all implemented during Phase 2, and OpenDDS is now in compliance with the specification. In addition to this development, internal tuning was performed as well. This included implementation of publisher side filtering for the content filtering and transport auto-selection, which simplifies and extends the previous transport specification capabilities. We also extended the development work to begin implementation of the interoperability transport specification, RTPS [Ref. 9.], for use both as a transport and a discovery mechanism.

The object model profile includes a Data Local Reconstruction Layer (DLRL) that is not implemented by OpenDDS. It requires implementation of an optional feature to permit coherent communications for grouping of data. This implements the PRESENTATION policy *access_scope* value of 'GROUP', allowing use of the DCPS API for grouping delivery of data. This allows applications the ability to implement object replication by guaranteeing the coherency of entire sets of data.

The ownership profile extends support of the OWNERSHIP policy values to include *kind*=='EXCLUSIVE'; implementing the OWNERSHIP_STRENGTH policy; and, adding the ability to set values for the HISTORY policy *depth* values to greater than 1. This allows applications to implement '*hot sparing*' style redundancy by setting policy values (and providing the spare publications).

The content-subscription profile adds the *ContentFilteredTopic* and *MultiTopic* entities to the API as well as a *QueryCondition* object that can be used to restrict data available for reading. These entities and conditions allow applications the ability to specify data to be received through the use of a SQL query using a sub-set of SQL that is appropriate for a streaming data style read. The implementation includes a publisher side filtering feature that pushes the query and its parameters from the receivers to

the publishers so that if data will not be received on the remote end of an association, it will not be sent. This allows reduced network loading when using this profile.

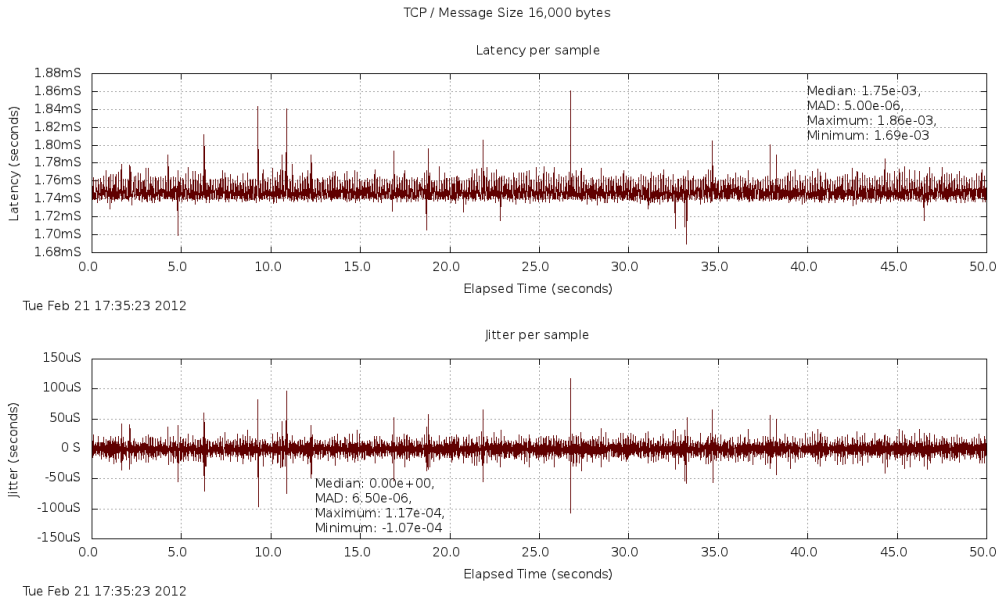
The internal transport API was simplified and extended to allow the use of defaults without programmer selections or configuration settings. Previously, the application program was required to establish the bindings between the actual transport implementation and the DCPS layer. As part of the Phase 2 development work, we simplified this so that if no specification was made that a default transport would be used along with its default settings. We also extended the configuration mechanism and data model to allow sets of transports to be specified in priority order. The priority ordered lists of transports for each end of an association are then matched to determine which transport implementation will be used. Where previously a best effort and reliable publication to different subscriptions would have required different writers, the transport layer can now provide the ability for a single writer to send data to both types of receivers.

The RTPS interoperability transport and discovery mechanisms were implemented through coordination with other projects using OpenDDS. This feature will allow OpenDDS to be integrated into legacy systems using other DDS implementations.

4.4 Performance Characterization

During the development activities of Phase 2, we continued executing the performance testing that was developed during Phase 1. This ensured that we did not adversely impact performance of the OpenDDS implementation as we modified and extended the features. This testing included both throughput and latency testing. The final performance is not significantly different from that at the beginning of Phase 2. The performance testing was done in an environment that reflects the ESTEL laboratories with a 100 Mbps network connection. The available bandwidth was determined by executing data transfers with ftp between hosts and was determined to be 875 Kbps. The throughput testing was done to require capacities up to 2 Gbps and showed all capacity was utilized for tests requiring more than what was available, i.e. there was no limiting effect (foldback) for rates overdriving the available capacity.

Latency and jitter data were collected for various payload sizes, sent at a rate of 100 Hz to not stress the network bandwidth limitations. A representative plot of the raw measurements is shown in Drawing 35. This plot includes both latency and jitter measurements with robust statistics for the data annotated on the chart. The tests were executed for 120 seconds, with the last 5,000 measurements retained and plotted. The median, median absolute deviation (MAD) and extremes noted on the charts reflect the statistics for the entire 120 second test execution. Since the test was executed without privileges on a time shared system, the data has noise that would not be present in a real-time scheduled system.

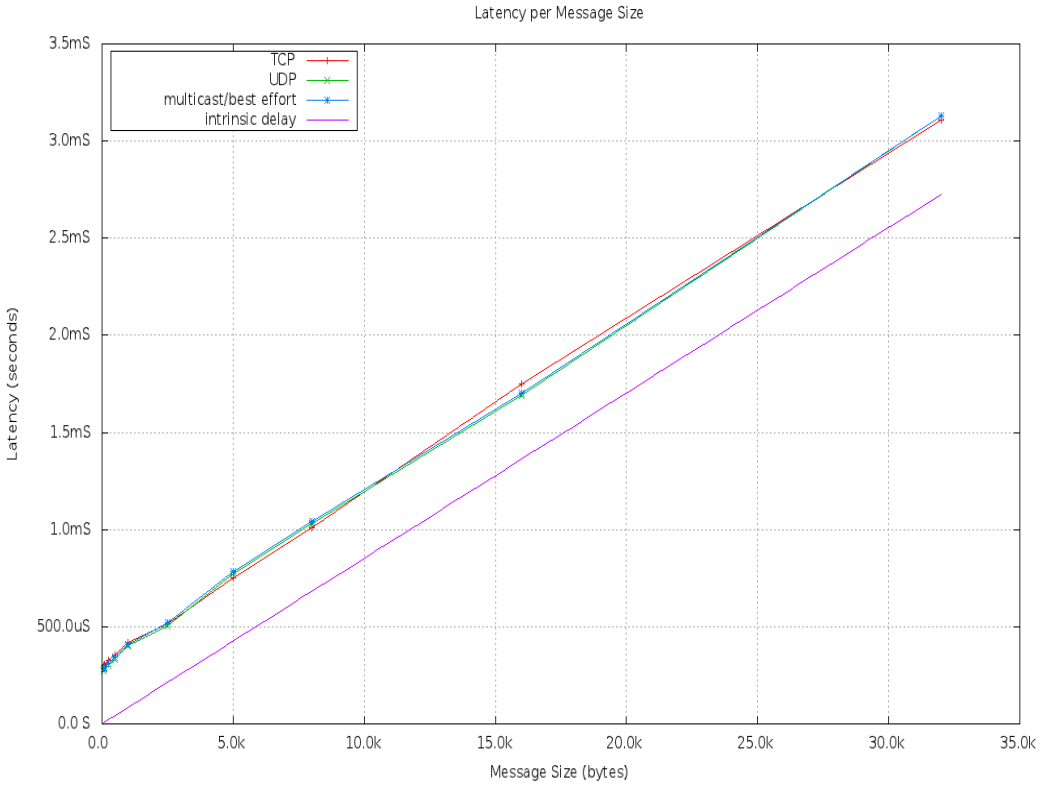


Drawing 35: Performance - Measurements

The latency measurements were taken for various payload sample sizes for the different OpenDDS transport implementations. This is charted in Drawing 36. The intrinsic (transport) delay shows the time for the data to be received over the test network with no latency at all. This is an artifact of the measurement; which is from the time the data is written to the time the entire data sample is read at the remote end. The plotted latency shows that the delay through the OpenDDS layer is consistent across payload sizes.

The plot also shows that the performance is independent of the transport mechanism, so other considerations such as reliability and number of destinations should be used to select the

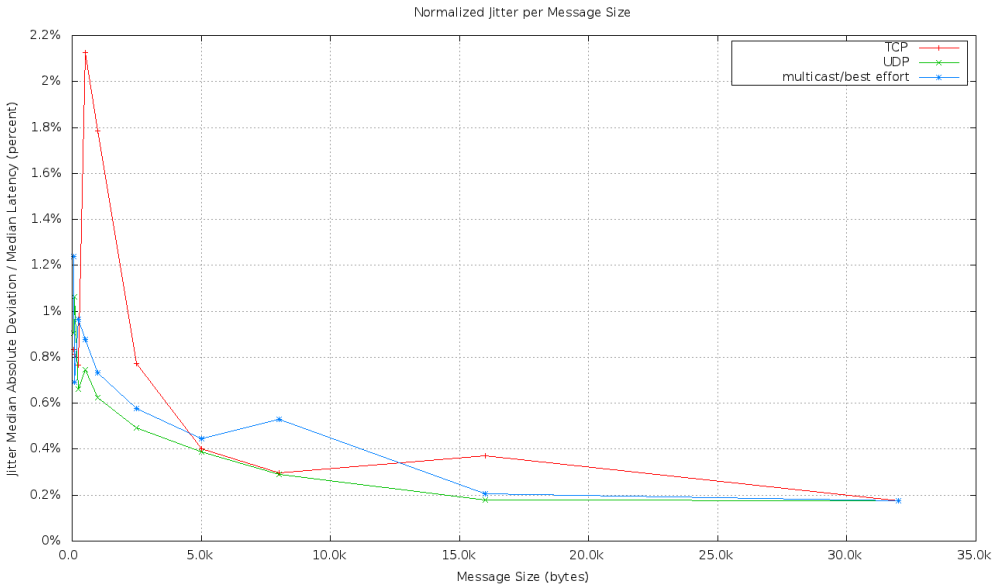
implementation to use for a given application. The testing was done for different sizes of payload and measurements taken for a round trip, with the resulting latency referenced to a single hop by dividing the measured value by 2. The effects this has on the measurements is addressed in [Ref. 15.], which discusses how to interpret the performance measurements.



Drawing 36: Performance - Latency

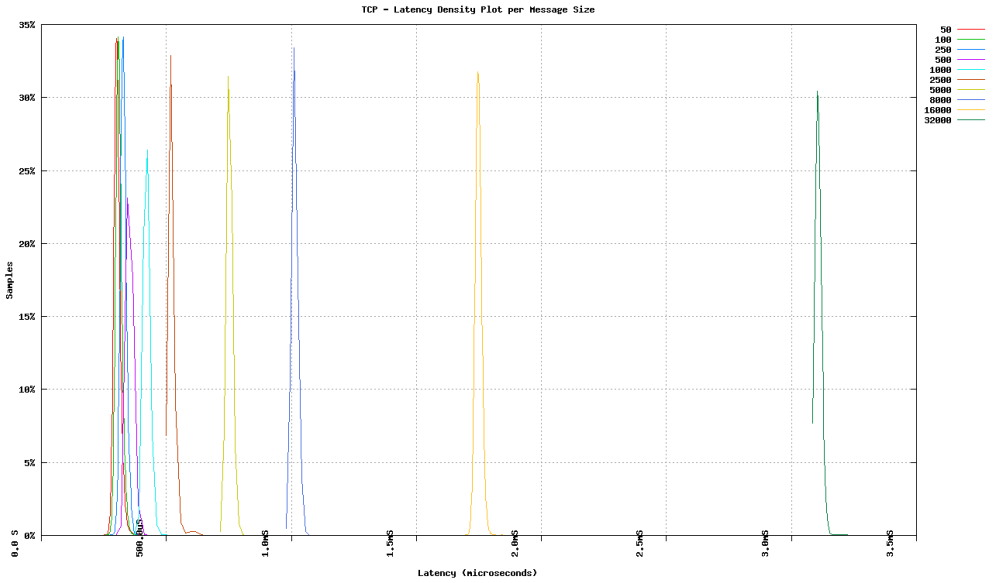
Jitter was measured as defined by RFC 3393 [Ref 16.], using the definition in §4.5 “Type-P-One-way-ipdv-jitter”. These values are calculated from the derived single-hop latency measurements described above. A plot of the measured jitter for various transport implementations over different payload sizes is shown in Drawing 37. The jitter is normalized for the expected delay and plotted as a percentage of the latency for the payload size. This allows meaningful comparisons to be made for the jitter at the different message sizes. The smaller payloads have a larger jitter value, which is expected since the overhead for sending the payload is a larger portion of the total transport time and is expected to be more variable than the payload between messages. Other than the single measurement of the TCP

transport at 500 bytes having a measured jitter of approximately 2.1% of the latency, the remainder are at around 1% or less, with the jitter measured as less than 0.5% for payloads of 8K bytes and larger.

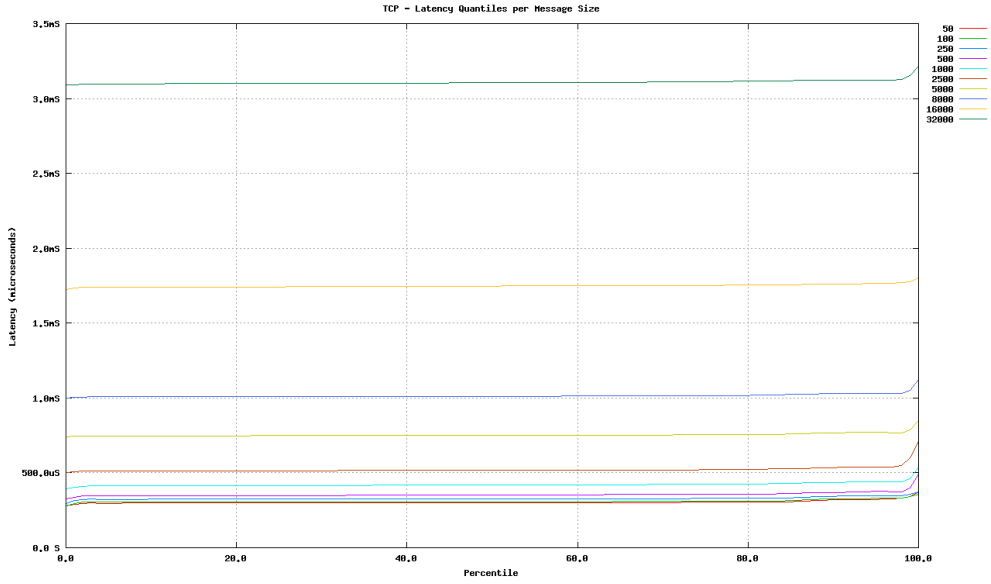


Drawing 37: Performance - Jitter

In addition to the aggregate statistics shown above, the latency delay for each sample was analyzed. The results of this analysis are grouped for each transport and can be understood with two charts: one that displays the latency delays as a density function plotted for each payload size as in Drawing 38; and one that displays the latency delays as quantiles plotted for each payload size as in Drawing 39. The density functions provide information about the spread, peakiness, and asymmetry of transporting each size payload. The quantile plots show the same information as a cumulative density function (rotated) that makes any anomalies in the latency performance explicit. A detailed description of these charts and their meaning can be found in [Ref. 15.].



Drawing 38: Performance - Latency Density



Drawing 39: Performance - Latency Quantiles

5 Recommendations for Further Development

On completion of Phase 2 development, there are several areas where the OpenDDS product could be extended with additional capabilities and features. These can be segregated into possible Phase 2.5 development, Phase 3 development, and additional commercial applications. This extended development includes implementation of higher layer abstractions, interoperability, and security features. These are summarized in the table below.

Category	Feature	Description
Phase 2.5	Interoperability	Implement RTPS interoperability specification as a transport implementation for OpenDDS.
		Implement RTPS discovery protocol for interoperability with other DDS implementations.
	Shared Memory	Implement a shared memory transport mechanism to increase collocated performance.
	Future Airborne Capabilities Environment (FACE) integration	Implement the FACE higher layer abstraction API wrapping the DDS API to allow OpenDDS to be used within a FACE system.
	Data Local Reconstruction Layer (DLRL)	Implement the DLRL abstraction layer for OpenDDS to simplify the application programming mechanisms.
	DLRL SDK	Extend the modeling toolkit to include the DLRL elements and features.
	Monitoring Tools	Extend the existing OpenDDS service monitoring tool to provide more dynamic information.
		Integrate OpenDDS with existing system management and monitoring tools such as Nagios and Hyperic.
	Libraries	Generate IDL libraries of defined DoD messages (such as Link11) to be available for development work with OpenDDS.
Phase 3	Scalability	Extend the scalability of OpenDDS to larger domains to allow use in broader applications. This includes scaling the number of topics and instances as well as the amount of data and participants during operation.
		Test and improve the scalability of the reliable multi-cast implementation for OpenDDS.

Category	Feature	Description
	Recording service	Implement a feature to allow capture and playback of message streams generated by OpenDDS for review and analysis. This enables audit trails of service operation to be generated as well.
	API extensions	Implement the C++11 language mappings as they are formalized by the OMG.
		Implement self describing topic features as they are formalized by the OMG.
	Web services	Research and develop a web service integrated implementation for OpenDDS.
	Topic level security	Implement Topic level security capabilities for OpenDDS. This is to include key management and a CA as well as implementation of secure transport layers. This is not necessarily conformant to the proposed (preliminary) security specifications from the OMG.
Other	Small footprint	Implement a small footprint version of OpenDDS suitable for use in embedded systems.
	Static discovery	Implement static discovery mechanisms to support small footprint and embedded system applications.
	CORBA Component Model	Integrate OpenDDS with the DDS OMG Corba Component Model (DDS4CCM) abstraction.
	Delay Tolerant Networking	Implement an RFC5050 DTN capability, including bundle processing and a convergence layer.
	Federated security	Extend Topic level security to be supported with the use of OpenDDS federated meta-data repositories.

6 References

1. Doxygen documentation generator,
   http://www.doxygen.org/
2. Eclipse Graphical Modeling Project
   http://www.eclipse.org/modeling/gmp/
3. Eclipse Modeling Framework project
   http://www.eclipse.org/modeling/emf/?project=emf
4. Future Airborne Capability Environment (FACE™)
   https://www.opengroup.us/face/
5. GraphViz – Graphical Visualization Software
   http://www.graphviz.org/
6. Makefile, Project, and Workspace creator tool,
   http://www.ociweb.com/products/MPC
7. Object Management Group, “Data Distribution Service for Real-time Systems,” Version 1.2, OMG Document formal/07-01-01, January 2007,
   http://www.omg.org/cgi-bin/doc?formal/07-01-01
8. Object Management Group, “OMG's Meta Object Facility”
   http://www.omg.org/mof/
9. Object Management Group, “The Real-time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol Specification ,” Version 2.1, OMG Document formal/09-05-01, January 2009,
   http://www.omg.org/spec/DDSI/2.1
10. Object Management Group, “UML Profile for Data Distribution Service,” OMG Document ptc/08-07-02, June 2008,
    http://www.omg.org/cgi-bin/doc?ptc/2008-07-02
11. Object Management Group, “XML Metadata Interchange (XMI),” Version 2.1.1, OMG Document formal/07-12-01, December 2007.
    http://www.omg.org/cgi-bin/doc?formal/07-12-01
12. OpenDDS Modeling Toolkit Eclipse installation site,
    http://www.opendds.org/modeling/eclipse/
13. OpenDDS Portal,
    http://www.opendds.org/
14. OpenDDS subversion source code repository,
    svn://svn.dre.vanderbilt.edu/DOC/DDS/trunk

15. Interpreting Performance Test Measurements,
http://mnb.ociweb.com/mnb/MiddlewareNewsBrief-201003.html

16. RFC 3393, “IP Packet Delay Variation Metric for IP Performance Metrics (IPPM)”,
http://tools.ietf.org/html/rfc3393

17. Wireshark, network protocol analyzer.
http://www.wireshark.org/